ARTÍCULOS DE TECNOLOGÍA

Creando anotaciones en Java

M

mario-alvial

30/09/2022

Tenemos la siguiente clase que representa a un usuario en nuestro sistema:

```
public class Usuario {

    private String nombre;
    private String identidad;
    private LocalDate fechaNacimiento;
}
```

Para guardar un nuevo usuario, se realizan varias validaciones, como por ejemplo: Ver si el nombre solo contiene letras, la IDENTIDAD solo números y ver si el usuario tiene al menos 18 años. Aquí está el método que hace esta validación:

```
public boolean usuarioValido(Usuario usuario){
    if(!usuario.getNombre().matches("[a-zA-Záàâãéèêíïóôõöúçñ\\s]+")){
        return false;
    }
    if(!usuario.getIdentidad().matches("[^0-9]+")){
        return false;
    }
    return Period.between(usuario.getFechaNacimiento(), LocalDate.now()).getYea
}
```

Supongamos ahora que tengo otra clase, la clase `Producto`, que contiene un atributo nombre y quiero hacer la misma validación que hice para el nombre de usuario: Ver si sólo contiene letras. ¿Y entonces? ¿Voy a crear otro método para hacer la misma validación? ¿O crear una interfaz o una clase que tanto Usuario como Producto extienden? No tiene mucho sentido ¿verdad? ¿Cómo resolver este caso sin repetir código?

Anotaciones

En Java 5 se ha introducido un nuevo recurso al lenguaje, las anotaciones. Permiten que los metadatos se escriban directamente en el código.

Los metadatos son, por definición, datos que hacen referencia a los propios datos.

> *En el contexto de la orientación a objetos, los metadatos son informaciones sobre los elementos del código. Esta información se puede definir en cualquier medio, solo con*

> *que el software o componente la recupere y la utilice para agregar nueva información en los elementos del código.*

Ten en cuenta que las notas por sí solas no hacen nada. Necesitan que la aplicación las recupere y las utilice para que solo así puedan proporcionarnos algo que podamos usar para realizar alguna tarea.

Volviendo a nuestro problema, vamos a crear una anotación para validar la edad mínima del usuario. Para ello, vamos a anotar nuestra clase:

```
public class Usuario {

    private String nombre;
    private String identidad;
    @EdadMinima
    private LocalDate fechaNacimiento;
```

Si miramos nuestro código nos daremos cuenta de que no compila, ya que falta implementar la anotación `@EdadMinina`. Por lo tanto, necesitamos crear una nueva clase con el nombre IdadeMinima:

```
public class EdadMinima {
}
```

Pero, pensándolo bien, ¿estamos creando una clase? ¡No lo estamos! Por lo tanto, la nomenclatura es diferente para una anotación. La forma correcta sería:

```
public @interface EdadMinima {
}
```

Extraño, ¿no? Pero fue la manera que la gente de Java hizo para decir que ese archivo es una anotación.

Ahora tenemos que anotar nuestra interfaz con algunas anotaciones obligatorias para que Java entienda dónde y cuándo se puede utilizar su anotación, siendo ellas:

- `@Retention` - Aquí hablaremos de nuestra aplicación hasta que nuestra anotación esté disponible.

- `@Target` - Aquí pasaremos los elementos que se pueden anotar con esta anotación.

¿Hasta dónde estará disponible nuestra anotación? Necesitamos que se ejecute cuando el usuario envíe sus datos, y esto sucede cuando nuestra aplicación se está ejecutando, por lo que la necesitamos en tiempo de ejecución, *Runtime*:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface EdadMinima {
}
```

¿Y quién será anotado? ¿Qué elemento tiene sentido ser anotado con una anotación que verifica si el usuario tiene edad suficiente? ¿Un atributo, cierto? Luego, un *Field*:

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface EdadMinima {
}
```

Ahora que hemos especificado el contexto de nuestra anotación, tenemos que hablar de la edad mínima que nuestra anotación debe utilizar para validar la edad del usuario, para ello, vamos a crear una propiedad en nuestra anotación llamada `valor`:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface EdadMinima {
    int valor();
}
```

Nuestra anotación está completa, ahora anotemos el atributo `fechaNacimiento` de la clase `Usuario` con ella:

```
public class Usuario {

    private String nombre;
    private String identidad;
    @EdadMinima
    private LocalDate fechaNacimiento;
```

```
    //getters e setters
}
```

Solo haciendo eso, recibiremos un error de compilación. Necesitamos pasar la edad mínima para nuestra anotación, luego:

```
public class Usuario {

    private String nombre;
    private String identidad;
    @EdadMinima(valor=18)
    private LocalDate fechaNacimiento;
    //getters e setters
}
```

Para validar esta anotación vamos a crear un usuario y pasar a un método `validador()`:

```
public static void main(String[] args) {
   Usuario usuario = new Usuario("Maria", "42198284863", LocalDate.of(1995, Mo
   System.out.println(validador(usuario));
}
```

Ahora vamos a crear el método `validador()` que devolverá un `boolean`:

```
public static boolean validador(Usuario usuario) {
}
```

El problema de crear nuestro método de esta manera es que nuevamente nosotros estamos limitandonos a validar solo usuarios, solo que nuestra meta es validar cualquier objeto.

Para ello, podemos hacer uso de *Generics* que, en nuestro caso, permitirá recibir un objeto de cualquier tipo:

```
public static <T> boolean validador(T objeto) {
}
```

Ahora estamos hablando de que vamos a recibir un objeto de tipo genérico `T`. Pero hacerlo no es suficiente, necesitamos validar ese objeto. ¿Y cómo validamos un objeto que no conocemos?

Necesitamos descubrir, en tiempo de ejecución, informaciones acerca del objeto que llegará en nuestro método, luego, podemos usar reflexión. Con reflexión podemos descubrir y operar datos de la clase. Entonces, primero, vamos a tomar la clase de ese objeto:

```
public static <T> boolean validador(T objeto) {
   Class<?> clase = objeto.getClass();
}
```

Con eso conseguimos operar la clase referente al tipo del objeto recibido. Primero, averigüemos qué atributo de nuestra clase está anotado con `@EdadMinima`.

Para averiguarlo, iteraremos sobre los atributos de la clase utilizando el método `getDeclaredFields()`:

```
public static <T> boolean validador(T objeto) {
   Class<?> clase = objeto.getClass();
   for (Field field : clase.getDeclaredFields()) {
   }
}
```

Ahora estamos iterando por todos los atributos de clase de nuestro objeto. El siguiente paso es averiguar qué campo está anotado con nuestra anotación.

Usaremos el método isAnnotationPresent(). Este método comprueba si el campo contiene la anotación pasada y devuelve un `boolean`.

```
public static <T> boolean validador(T objeto) {
   Class<?> clase = objeto.getClass();
   for (Field field : clase.getDeclaredFields()) {
      if (field.isAnnotationPresent(EdadMinima.class)) {
```

```
        }
    }
}
```

Si entra en el `if` sabremos que el campo tiene la anotación `EdadMinima`. Solo falta comparar la edad mínima que asignamos en nuestra anotación con la edad pasada. Para hacer eso vamos a recoger nuestra anotación.

Para conseguir nuestra anotación usaremos el método getAnnotation() pasando nuestra anotación:

```
public static <T> boolean validador(T objeto) {
    Class<?> clase = objeto.getClass();
    for (Field field : clase.getDeclaredFields()) {
        if (field.isAnnotationPresent(EdadMinima.class)) {
            EdadMinima edadMinima = field.getAnnotation(EdadMinima.class);
        }
    }
}
```

Tenemos un objeto del tipo de nuestra anotación, con él conseguimos coger la edad mínima que setamos. Ahora necesitamos, también, la edad pasada por el usuario.

Para acceder al valor de un atributo `private` necesitamos decir que este atributo está accesible de esta manera:

```
public static <T> boolean validador(T objeto) {
    Class<?> clase = objeto.getClass();
    for (Field field : clase.getDeclaredFields()) {
        if (field.isAnnotationPresent(EdadMinima.class)) {
            EdadMinima edadMinima = field.getAnnotation(EdadMinima.class);
            field.setAccessible(true);
        }
```

```
    }
}
```

Pero tenga en cuenta que no recibimos la edad del usuario, recibimos su fecha de nacimiento. Tenemos que tomar esta fecha y averiguar la edad del usuario. Para obtener el valor del atributo anotado con `@EdadMinima` usaremos el método get() de la clase `Field`:

```
public static <T> boolean validador(T objeto) {
    Class<?> clase = objeto.getClass();
    for (Field field : clase.getDeclaredFields()) {
        if (field.isAnnotationPresent(EdadMinima.class)) {
            EdadMinima edadMinima = field.getAnnotation(EdadMinima.class);
            try{
                field.setAccessible(true);
                LocalDate fechaNacimiento = (LocalDate) field.get(objeto);
             } catch (IllegalAccessException e) {
                  e.printStackTrace();
             }
        }
    }
}
```

Tenga en cuenta que hemos hecho un cast para `LocalDate`, ya que el método `get()` nos devuelve un `objeto`.

Para finalizar, vamos a comparar para ver si el período entre `fechaNacimiento` y la fecha actual es mayor o igual al valor que colocamos como edad mínima en nuestra anotación:

En la comparación usaremos el método between() que toma como parámetro dos fechas para ser comparadas, el método now() para obtener la fecha actual y el método getYears() para saber el valor del período en años:

```
public static <T> boolean validador(T objeto) {
    Class<?> clase = objeto.getClass();
    for (Field field : clase.getDeclaredFields()) {
        if (field.isAnnotationPresent(EdadMinima.class)) {
```

```
            EdadMinima edadMinima = field.getAnnotation(EdadMinima.class);
            try{
                field.setAccessible(true);
                LocalDate fechaNacimiento = (LocalDate) field.get(objeto);
                return Period.between(fechaNacimiento, LocalDate.now()).getYears()
             } catch (IllegalAccessException e) {
                  e.printStackTrace();
             }
        }
    }
    return false;
}
```

Tenga en cuenta que tomamos el período en años y comparamos para ver si el valor es mayor o igual a `EdadMinina.valor()` que no es más que el valor que ponemos en nuestra anotación. Además, al final volvemos falso, ya que si ningún campo del objeto posee la anotación `@IdadeMinima` no puede ser validado.

```
public static void main(String[] args) {
    Usuario usuario = new Usuario("Maria", "52902488033", LocalDate.of(2005, Mo
    System.out.println(validador(usuario));
}
```

La fecha de nacimiento del usuario creado es 13/01/2005, por lo que comparado con la fecha actual resultará en false. Vamos a ver:

```
false

Process finished with exit code 0
```

Ahora vamos a probar con la fecha 10/01/2000:

```
public static void main(String[] args) {
    Usuario usuario = new Usuario("Maria", "52902488033", LocalDate.of(2000, Mo
```

```
    System.out.println(validador(usuario));
}
```

¡Bien! Ahora nuestro `validador()` puede validar cualquier clase que tenga nuestra anotación.

Conclusión

En este post podemos descubrir el poder de la reflexión, realmente nos ayuda y mucho cuando necesitamos operar sobre la clase de los objetos dinámicamente.

Con anotaciones hemos sido capaces de marcar los elementos de nuestra aplicación para que nuestro método que usa reflexión logre captar informaciones útiles para que fuese posible ejecutar nuestra lógica de validación.

Usar la reflexión es muy útil cuando queremos crear algo más genérico, pero debemos tener cuidado porque con la reflexión, operamos sobre los tipos de objetos dinámicamente y esto hace que algunas optimizaciones de la máquina virtual no se ejecuten. Así que tenemos que tener cuidado de ver dónde es realmente necesario el uso de la reflexión.

Artículo escrito por Mario Alvial

Este articulo fue adecuado para Alura Latam por: Adriana Oliveira

ARTÍCULOS DE TECNOLOGÍA

ALIADOS

Empresa participante do SCALE UP ENDEAVOR

En Alura somos unas de las Scale-Ups seleccionadas por Endeavor, programa de aceleración de las empresas que más crecen en el país.

Google for Startups Alumni 2021

Fuimos unas de las 7 startups seleccionadas por Google For Startups en participar del programa Growth Academy en 2021

POWERED BY

CURSOS

Cursos de Programación
Lógica de Programación | Java

Cursos de Front End
HTML y CSS | JavaScript | React

Cursos de Data Science
Data Science | Machine Learning | Excel | Base de Datos | Data Visualization | Estadística

Cursos de DevOps
Docker | Linux

Cursos de Innovación y Gestión